**Capítulo 3**

**Solução Proposta**

Neste capítulo será apresentado inicialmente o InteropFrame em sua implementação atual. Em seguida será apresentada a solução proposta para lidar com as limitações do InteropFrame, além de uma proposta de extensão para o suporte ao modelo de componentes OSGi dentro da ferramenta.

3.1 InteropFrame

Segundo {nascimento2013}, o papel do *framework* InteropFrame é o de prover uma solução de *binding* (interconexão) transparente entre componentes distribuídos de modelos diferentes. Ele possibilita que os componentes envolvidos na construção de aplicações distribuídas interajam através dos mecanismos de interoperabilidade providos pelo *framework*.

Com o InteropFrame é possível tornar interoperáveis sistemas desenvolvidos nos modelos de componentes OpenCOM e Fractal. Esta interoperabilidade pode ocorrer tanto em sistemas locais, como também com partes distribuídas pela rede. O *binding* remoto entre os componentes é suportado através dos mecanismos Java RMI e Web Services SOAP.

A Figura {figuras/interopframe.png} apresenta a arquitetura do InteropFrame. A seguir são explicados os módulos dessa arquitetura {nascimento2013}:

- Configurador Distribuído (CD) – módulo responsável pelo gerenciamento do serviço de interoperabilidade entre os componentes distribuídos. Este módulo coordena e controla as operações dos demais módulos do framework distribuído;

- Plug-ins de Modelos de Componentes (PMC) – cada plug-in permite que o InteropFrame suporte um modelo de componentes específico. Um Plug-in de modelo de componente é composto pelos seguintes submódulos:

- Gerador de *Proxies* (GP) – responsável pela geração automática dos *proxies* que possibilitam a interoperabilidade entre os componentes distribuídos de modelos diferentes. Este módulo baseia-se na geração de código a partir de *templates* pré-definidos para cada modelo de componentes específico;

- Montador de *Proxies* (MP) - responsável pela execução sob demanda dos *proxies* criados pelo GP, bem como a disponibilização do serviço de interoperabilidade entre os componentes distribuídos;

- Repositório de *Proxies* (RP) – repositório para armazenamento dos *proxies* gerados pelo GP.

- Plug-ins de Geradores de *Bindings* (GB) - cada plug-in gerador de *binding* é responsável pela geração automática do código-fonte de um tipo diferente de *binding* entre componentes remotos. O módulo Gerador de *Proxies* faz uso de um tipo específico de Gerador de *Binding* para enxertar nos *proxies* o código que promove a interconexão remota.



Figura {figuras/interopframe.png} – Arquitetura do InteropFrame {nascimento2013}.

A Figura {figuras/interopframe02.png} mostra o funcionamento do InteropFrame. Neste cenário o usuário deseja interconectar os componentes “A” e “B”, desenvolvidos respectivamente em OpenCOM e Fractal. O componente “A” é tratado aqui como componente cliente pois requisita os serviços do componente servidor “B” de modo remoto.

O detalhamento deste processo de funcionamento para o exemplo da Figura {figuras/interopframe02.png} é descrito a seguir {nascimento2013}:

- Do lado do componente cliente

(1C) O Configurador Distribuído (CD) verifica se o *proxy* do lado cliente, necessário para promover a interoperabilidade, já se encontra no Repositório de *Proxies* (RP), caso contrário ele solicita a geração do mesmo no passo 2C. Caso o componente *proxy* já exista, o próximo passo será o 5C, onde esse componente será utilizado pelo Montador de *Proxies* (MP);

(2C) O CD solicita ao submódulo Gerador de Proxies (GP) do modelo de componentes OpenCOM para gerar automaticamente o código do componente “X” que representa o *proxy* do lado cliente;

(3C) O GP solicita ao submódulo Gerador de *bindings* (GB) do RMI para gerar automaticamente a parte do código do componente “X” responsável pela comunicação remota;

(4C) O GP armazena no RP o componente “X” gerado;

(5C) O CD solicita ao MP que proceda com a inicialização do *proxy* do lado cliente;

(6C) O MP obtém e inicializa o componente “X” do lado cliente no ambiente de execução OpenCOM conectando o receptáculo do componente “A” à interface provida do *proxy* “X”. O *proxy* “X” do lado cliente representa o componente “B” no lado cliente e tem seus serviços requisitados pelo componente “A”.

- Do lado do componente servidor

(1S) O Configurador Distribuído (CD) verifica se o *proxy* do lado servidor já se encontra no Repositório de *Proxies* (RP), caso contrário ele solicita a geração do mesmo no passo 2S. Caso o componente *proxy* já exista, o próximo passo será o 5S, onde esse componente será utilizado pelo Montador de *Proxies* (MP);

(2S) O CD solicita ao submódulo GP do modelo de componentes Fractal para gerar automaticamente o código do componente “Y” que representa o *proxy* (também chamado de *skeleton*) do lado servidor;

(3S) O GP solicita ao submódulo GB do RMI para gerar automaticamente a parte do código do componente “Y” responsável pela comunicação remota;

(4S) O GP armazena no RP o componente “Y” gerado;

(5S) O CD solicita ao MP que proceda com a inicialização do *proxy* do lado servidor;

(6S) O MP obtém e inicializa o componente “Y” do lado servidor no ambiente de execução Fractal conectando o receptáculo do *proxy* “Y” à interface provida do componente “B”. O *proxy* “Y” do lado servidor representa o componente “A” no lado servidor que requisita os serviços do componente “B”.



Figura {figuras/interopframe02.png} – Funcionamento do InteropFrame {nascimento2013}.

Com o *binding* executado, o componente “A” agora pode utilizar os serviços do componente “B” de forma transparente. Quando uma requisição é feita no componente “A” ela é repassada via RMI do componente *proxy* “X” para o componente *skeleton* “Y” e este por sua vez repassa para o componente “B”. A resposta dessa requisição é feita pelo caminho inverso, de “B” para “Y”, de “Y” para “X” e de “X” para “A”. Na prática, “A” e “X” são componentes do modelo OpenCOM, assim como “B” e “Y” são do modelo Fractal. Os componentes “X” e “Y” se comunicam através de RMI, garantindo assim a interoperabilidade entre os componentes “A” e “B”.

3.2 Modularização do InteropFrame

O InteropFrame foi desenvolvido em Java “puro”, de forma a permitir a extensibilidade para novos modelos de componentes e de *bindings*. Cada plug-in de modelo de componentes ou de *binding* fornece o suporte a um modelo de componentes ou de *binding* específico. Com o desenvolvimento de novos plug-ins a ferramenta passa a suportar novos modelos.

Segundo {hall2011} o Java provê alguns aspectos de modularização através da orientação a objetos, porém não foi proposto para suportar modularização de alta granularidade. {hall2011} ainda cita algumas limitações do Java no quesito modularização:

- Baixo nível de controle de visibilidade de código: Os modificadores de acesso do Java (*public, protected* e *private*) tratam em baixo nível o encapsulamento da orientação a objetos e não no nível de particionamento lógico do sistema. Em Java, um *package* (pacote) é tipicamente utilizado para particionar código. Para este código ser visível por um outro *package*, ele deve ser declarado como *public*. Algumas vezes, a estrutura lógica da aplicação faz chamadas a códigos de *packages* diferentes, significando que qualquer dependência entre os pacotes deve ser exposta como *public*. Dessa maneira, os detalhes de implementação tornam-se públicos, dificultando a evolução do sistema devido a possível criação de dependências da API não pública.

- Conceito de *Classpath* propenso a erros: Aplicações são compostas de várias versões de bibliotecas e componentes. O *Classpath* do Java não lida com versões de código, retornando assim o primeiro que encontra. O modo de construção do *Classpath* não permite especificar versões de um mesmo código. Em Java apenas se vai colocando as bibliotecas (comumente arquivos JAR) até que a JVM (*Java Virtual Machine*) pare de acusar erros sobre classes faltantes.

- Implantação limitada e suporte a gerenciamento: Não há maneira fácil em Java de se implantar um conjunto particular de dependências de código versionadas e executar a aplicação. Também é dificultada a evolução da aplicação e seus componentes após a implantação. O Java não possui um suporte direto à criação de plug-ins dinâmicos, o que é conseguido apenas através do uso de *Class Loaders* – mecanismos de baixo nível e propensos a erros.

Tendo em vista as limitações do Java, necessita-se de uma maneira mais eficiente para a modularização do InteropFrame. A proposta deste trabalho consiste na adoção do OSGi como mecanismo de modularização. A extensibilidade do InteropFrame passará a ter um suporte facilitado, uma vez que cada Plug-in de Modelo de Componente e seus respectivos submódulos seriam portados para *bundles* OSGi independentes. Da mesma forma, cada Gerador de Binding também se tornaria um *bundle* independente. O do núcleo do InteropFrame, bem como o Configurador Distribuído também comporiam novos *bundles*.

3.3 Solução de comunicação do Configurador Distribuído

O Configurador Distribuído, módulo responsável pelo gerenciamento entre as partes distribuídas do InteropFrame, é desenvolvido utilizando a tecnologia Java RMI. O Configurador Distribuído atua na comunicação remota entre os lados servidor e cliente do InteropFrame. O lado servidor é responsável por propagar pela rede uma interface provida de um componente de um dado sistema. O lado cliente faz a utilização desse serviço fornecido pelo lado servidor.

Para garantir uma comunicação menos limitada que o RMI, este trabalho propõe

3.4 Extensão para o modelo de componentes OSGi

Além da modularização utilizando o OSGi, também é proposto neste trabalho a extensão para o suporte ao modelo de componentes OSGi dentro do InteropFrame